Two programs one memory

main() {
    ...
    yield()
}

main() {
    ...
    yield()
}

Save/restore
Or more like renting a set of rooms in an office building.
Or more like renting a set of rooms in an office building
Relocation

• One way to achieve this is to relocate program at different addresses
  • Remember relocation (from linking and loading)
Relocate binaries to work at different addresses
• One way to achieve this is to relocate program at different addresses

• What is the problem?
• One way to achieve this is to relocate program at different addresses

• What is the problem?
  • No isolation
Another way is to ask for hardware support
This is called segmentation
What are we aiming for?

- Illusion of a private address space
  - Identical copy of an address space in multiple programs
    - Remember `fork()`?
  - Simplifies software architecture
    - One program is not restricted by the memory layout of the others
Two processes, one memory?
Two processes, one memory?

- We want hardware to add base value to every address used in the program
Seems easy

- One problem
  - Where does this base address come from?
Seems easy

• One problem
  • Where does this base address come from?
  • Hardware can maintain a table of base addresses
    – One base for each process
  • Dedicate a special register to keep an index into that table
- One problem
  - Where does this base address come from?
  - Hardware can maintain a table of base addresses
    - One base for each process
  - Dedicate a special register to keep an index into that table
New addressing mode
All addresses are logical address

- They consist of two parts
  - Segment selector (16 bit) + offset (32 bit)
• Segment selector (16 bit)
  • Is simply an index into an array (Descriptor Table)
  • That holds segment descriptors
    - Base and limit (size) for each segment
Elements of the descriptor table are segment descriptors

- Base address
  - 0 – 4 GB
- Limit (size)
  - 0 – 4 GB
- Access rights
  - Executable, readable, writable
  - Privilege level (0 - 3)
• Offsets into segments (x in our example) or “Effective addresses” are in registers
Logical addresses are translated into physical addresses:

- Effective address + DescriptorTable[selector].Base
• Logical addresses are translated into physical
  • Effective address + DescriptorTable[selector].Base
• Logical addresses are translated into physical
• Effective address + DescriptorTable[selector].Base
- Logical addresses are translated into physical
- Effective address + DescriptorTable[selector].Base
- **Physical address =**
  
  \[ \text{Effective address} + \text{DescriptorTable}[\text{selector}].\text{Base} \]

- Effective addresses (or offsets) are in registers
- **Selector is in a special register**
• Offsets (effective addresses) are in registers
  • *Effective address + DescriptorTable[selector].Base*
  • But where is the selector?
Segment registers

- Hold 16 bit segment selectors
  - Pointers into a special table
  - Global or local descriptor table
- Segments are associated with one of three types of storage
  - Code
  - Data
  - Stack
Segmented programming (not real)

static int x = 1;
int y; // stack

if (x) {
    y = 1;
    printf ("Boo");
} else
    y = 0;

ds:x = 1; // data
ss:y;     // stack

if (ds:x) {
    ss:y = 1;
    cs:printf(ds:"Boo");
} else
    ss:y = 0;
Programming model

- Segments for: code, data, stack, “extra”
  - A program can have up to 6 total segments
  - Segments identified by registers: cs, ds, ss, es, fs, gs

- Prefix all memory accesses with desired segment:
  - `mov eax, ds:0x80` (load offset 0x80 from data into eax)
  - `jmp cs:0xab8` (jump execution to code offset 0xab8)
  - `mov ss:0x40, ecx` (move ecx to stack offset 0x40)
Programming model, cont.

• This is cumbersome,

• Instead the idea is: infer code, data and stack segments from the instruction type:
  • Control-flow instructions use code segment (jump, call)
  • Stack management (push/pop) uses stack
  • Most loads/stores use data segment

• Extra segments (es, fs, gs) must be used explicitly
Code segment

- Code
  - CS register
  - EIP is an offset inside the segment stored in CS
- Can only be changed with
  - procedure calls,
  - interrupt handling, or
  - task switching
Data segment

• Data
  • DS, ES, FS, GS
  • 4 possible data segments can be used at the same time
Stack segment

- Stack
  - SS
- Can be loaded explicitly
  - OS can set up multiple stacks
  - Of course, only one is accessible at a time
Segmentation works for isolation, i.e., it does provide programs with illusion of private memory
Segmentation is ok... but
What if process needs more memory?
What if process needs more memory?

Process 1 (ls)

```
x
```

```c
malloc() = x + base_{p_1}
```

Process 2 (ls)

```
x
```

```c
base_{p_2}
```

Memory

```
x + base_{p_1}
```

```
x + base_{p_2}
```
You can move P2 in memory

Process 1 (ls)

```
malloc() = x
```

```
base_{p_1}
```

```
x + base_{p_1}
```

Memory

Process 2 (ls)

```
x
```

```
base_{p_2}
```

```
x + base_{p_2}
```

move P2 (copy it's memory)
Or even swap it out to disk
Problems with segments

• But it's inefficient
  • Relocating or swapping the entire process takes time

• Memory gets fragmented
  • There might be no space (gap) for the swapped out process to come in
  • Will have to swap out other processes
Paging
Pages

Process 1 (Is)

Process 2 (Is)

Memory

Page table
Level 1

Level 2
Paging idea

• Break up memory into 4096-byte chunks called pages
  • Modern hardware supports 2MB, 4MB, and 1GB pages
• Independently control mapping for each page of linear address space

• Compare with segmentation (single base + limit)
  • many more degrees of freedom
Page translation

- Linear Address
  - 31: Directory
  - 22-21: Table
  - 12-11: Offset

- Page Directory
  - 10: PDE with PS=0

- CR3
  - 32

- Page Table
  - 10

- PTE
  - 20

- 4-KByte Page
  - 12

- Physical Address
  - 20
Page directory entry (PDE)

<table>
<thead>
<tr>
<th>Address of page table</th>
<th>Ignored</th>
<th>0</th>
<th>Ign</th>
<th>A</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>P</th>
<th>W</th>
<th>U/S</th>
<th>R/S</th>
<th>W</th>
<th>PDE: page table</th>
</tr>
</thead>
</table>

- 20 bit address of the page table
Page directory entry (PDE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits
Page directory entry (PDE)

- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits

- Pages 4KB each, we need 1M to cover 4GB
- Pages start at 4KB (page aligned boundary)
Page directory entry (PDE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Address of page table | Ignored | O | Ign | A | P | C | D | P | W | U | S | R | W | 1 | PDE: page table |

- Bit #1: R/W – writes allowed?
- But allowed where?
Page directory entry (PDE)

- Bit #1: R/W – writes allowed?
  - But allowed where?
  - One page directory entry controls 1024 Level 2 page tables
    - Each Level 2 maps 4KB page
  - So it's a region of 4KB x 1024 = 4MB
Page directory entry (PDE)

- Bit #2: U/S – user/supervisor
  - If 0 – user-mode access is not allowed
  - Allows protecting kernel memory from user-level applications
### Page table entry (PTE)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Address of 4KB page frame</td>
</tr>
<tr>
<td>30</td>
<td>Ignored</td>
</tr>
<tr>
<td>29</td>
<td>G</td>
</tr>
<tr>
<td>28</td>
<td>P</td>
</tr>
<tr>
<td>27</td>
<td>T</td>
</tr>
<tr>
<td>26</td>
<td>A</td>
</tr>
<tr>
<td>25</td>
<td>D</td>
</tr>
<tr>
<td>24</td>
<td>P</td>
</tr>
<tr>
<td>23</td>
<td>C</td>
</tr>
<tr>
<td>22</td>
<td>D</td>
</tr>
<tr>
<td>21</td>
<td>Write bits</td>
</tr>
<tr>
<td>20</td>
<td>Bit #1: R/W – writes allowed?</td>
</tr>
<tr>
<td>19</td>
<td>Bit #2: U/S – user/supervisor</td>
</tr>
<tr>
<td>18</td>
<td>If 0 user-mode access is not allowed</td>
</tr>
<tr>
<td>17</td>
<td>Bit #5: A – accessed</td>
</tr>
<tr>
<td>16</td>
<td>Bit #6: D – dirty – software has written to this page</td>
</tr>
</tbody>
</table>

- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB
- Bit #1: R/W – writes allowed?
  - To a 4KB page
- Bit #2: U/S – user/supervisor
  - If 0 user-mode access is not allowed
- Bit #5: A – accessed
- Bit #6: D – dirty – software has written to this page
Page translation

Linear Address

31 22 21 12 11 0
Directory Table Offset

12 4-KByte Page

10 Page Directory

10 Page Table

PTE

20 Physical Address

32 CR3

PDE with PS=0
mov (%EBX), EAX  # move value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = 00 0000 0101|00 0000 0011|0000 0000 0001

Virtual Address Space (or Memory) of the Process

0 1 2

... page number

page number = 5123
or (0b1 0100 0000 0011)

Physical Memory

0 1 2 3 4 5 6 7 8 9 10 11 12

1M (1,048,575)
```assembly
mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = 00 0000 0101|00 0000 0011|0000 0000 0001

Virtual Address Space (or Memory) of the Process

CR3 = 0
0 1 2
0 1 2 3 4 5 6 7 8 9 10 11 12

page number = 5123
or (0b1 0100 0000 0011)

1M (1,048,575)
```
mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = \[00 0000 0101 00 0000 0011 0000 0000 0001\]

page number

CR3 = 0

Physical Memory

32 bits (4 bytes)

0 1 2 3 4 5 6 7 8 9 10 11 12

Level 1 (Page Table Directory)

page number = 5123
or (0b1 0100 0000 0011)

1M (1,048,575)
mov (%EBX), EAX  # mov value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = 00 0000 010 [00 0000 0011] 0000 0000 0001

CR3 = 0

1M (1,048,575)

page number = 5123
or (0b1 0100 0000 0011)

Physical Memory

32 bits (4 bytes)

Level 1 (Page Table Directory)

Level 2 (Page Table)
mov (%EBX), EAX  # move value from the location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809

20 983 809 = 00 0000 0101|00 0000 0011|0000 0000 0001

CR3 = 0

page number = 5123
or (0b1 0100 0000 0011)

1M (1,048,575)
• Result:
  • EAX = 55
But why do we need page tables

... Instead of arrays?

- Page tables represent sparse address space more efficiently
  - An entire array has to be allocated upfront
  - But if the address space uses a handful of pages
  - Only page tables (Level 1 and 2 need to be allocated to describe translation)
- On a dense address space this benefit goes away
  - I'll assign a homework!
What about isolation?

- Two programs, one memory?
What about isolation?

- Two programs, one memory?
- Each process has its own page table
- OS switches between them
P1 and P2 can't access each other memory
Compared to segments pages allow ...

- Emulate large virtual address space on a smaller physical memory
  - In our example we had only 12 physical pages
  - But the program can access all 1M pages in its 4GB address space
  - The OS will move other pages to disk
Compared to segments pages allow ...

- Share a region of memory across multiple programs
- Communication (shared buffer of messages)
- Shared libraries
More paging tricks

• Protect parts of the program
  • E.g., map code as read-only
    – Disable code modification attacks
    – Remember R/W bit in PTD/PTE entries!
  • E.g., map stack as non-executable
    – Protects from stack smashing attacks
    – Non-executable bit
Recap: complete address translation
Why do we need paging?

- Compared to segments pages provide fine-grained control over memory layout
  - No need to relocate/swap the entire segment
    - One page is enough
  - You're trading flexibility (granularity) for overhead of data structures required for translation
Questions?